Innovative Pedagogical Practices in the Craft of Computing

James H. Davenport, Alan Hayes, Rachid Hourizi

Department of Computer Science
University of Bath
Bath BA2 7AY, UK
Email: {J.H.Davenport,A.Hayes,R.Hourizi}@bath.ac.uk

Tom Crick

Department of Computing & Information Systems
Cardiff Metropolitan University
Cardiff CF5 2YB, UK
Email: tcrick@cardiffmet.ac.uk

***Abstract*—Teaching programming is much more like teaching a craft skill than it is an academic subject. Hence an "apprenticeship" model, where apprentices learn by watching the master do, and then do themselves, and are criticised in their doing, is, we claim, more appropriate than the "lecturer/lecturee" model that universities implicitly adopt. Furthermore, there are generally many more apprentices than the master can personally supervise. Universities will therefore use various tutors, who should be regarded as the analogue of the guild-master's journeymen. However, how does one encourage this mindset in students who, for their other courses, are indeed lecturees? What are the implications for the journeymen?**

1. Introduction: Problems and Contexts

Computer programming, the art of actually instructing a computer to do what one wants, is fundamentally a practical skill. How does one teach this practical skill in a university setting, to students who may not be initially motivated to acquire it, and who may have a variety of past experience, or none at all? How does one ensure that they progress to the rest of their studies with a firm background in programming, so that programming difficulties do not impede their other learning? Here experienced teachers from two different UK institutions describe how they have addressed these challenges. Above all, the key is to recognise that programming is a practical subject, and needs to be taught and assessed as such. A model we use is that of 'apprenticeship' [**?**], where the students learn how to do by seeing it done, and by being guided in their doing. Bath is a "top of the pile" institution, recruiting able students to both Mathematics and Computing degrees, but in neither case will they, in the current state of the UK curricula, have necessarily encountered any programming before entering university [**?**]. Cardiff Met offers degrees in computing, software engineering and business information systems to a broad range of students from diverse educational backgrounds. This paper focuses on Bath, but the same general techniques have been applied in Cardiff Met, to good effect there also. Hence the authors believe that their practice is worth disseminating.

At Bath, JHD teaches programming to 330 Mathematics students (Bath-M) in an innovative course [**?**], while RH teaches programming far more intensively to 120 Computer Science students (Bath-C). JHD has used this approach, refining it over the years, since the course started in 2009, while RH has been adopting parts of it as his course has grown from 70 to 120.

2. Programming Apprentices

Programming is a hard craft to master and its teaching is challenging. An apprentice model, where students learn their craft from a master is an approach that can lead to improved student engagement [**?**], [**?**]. Although traditionally applied to physical and vocational skills, the apprenticeship model can also be applied to the acquisition of cognitive skills such as those required for programming. In this context, the master is required to focus on the programming process and demonstrates it through writing, debugging and running 'live' programs. This takes place whilst being observed by the student cohort. Scaffolding is provided through the provision of regular practical exercises with good quality formative feedback.

This model is related to, but different from, the 'lab-first' approach discussed in [**?**]. Their own analysis (section 8.3) is worth considering: "*The lab-first approach has both advantages and disadvantages. On the one hand, in the spirit of constructivism, its main advantage is expressed by the active experience learners get in the computer lab, which in turn, establishes foundations based on which learners construct their mental image of the said topic; on the other hand, the lab-first teaching approach involves some insecurity feelings expressed both by the computer science teacher and the learners.*" In our case, insecurity among the learners is a real concern, and manifests itself very clearly among students who can arrive without any previous experience of programming and then find themselves in labs with relatively advanced programmers or miss the first two weeks of term and then start the course and its labs at a disadvantage. These late starters find their relative lack of experience daunting, despite the support provided by videos of past sessions. Constructivists interpret student learning as the development of personalised knowledge frameworks that are continually refined. According to this theory, to

learn, a student must actively construct knowledge, rather than simply absorbing it from textbooks and lectures [**?**]. Students develop their own self-constructed rules, or "alternative frameworks" [**?**]. For example, in programming, these alternative frameworks "naturally occur as part of the transfer and linking process" [**?**]; they represent the prior knowledge essential to the construction of new knowledge. When learning, the student modifies or expands his or her framework in order to incorporate new knowledge. We have also had an interest in threshold concepts [**?**], [**?**] — as a subset of the core concepts in a discipline — for computing, and more specifically, programming [**?**]. These are the building blocks that must be understood; in addition, they must be:

- *Transformative:* they change the way a student looks at things in the discipline;
- *Integrative:* they tie together concepts in ways that were previously unknown to the student;
- *Irreversible:* they are difficult for the student to unlearn;
- *Potentially troublesome for students:* they are conceptually difficult, alien, and/or counter-intuitive;
- *Often boundary markers:* they indicate the limits of a conceptual area or the discipline itself.

Students who have mastered these threshold concepts, however, have, at least in part, "crossed over from being outsiders to belonging to the field they are studying" [**?**], although there is some dispute on how they apply to computer science [**?**]. Our interest in these transformations, integrations, potentially troublesome challenges and boundaries informs the structure of the programming courses at both Bath and Cardiff Met, the work we expect from our students and the help that we require from our course tutors (the journeymen in our apprenticeship model). We introduce our approach to these areas in this short practice paper and will expand upon each one in future work.

3. Course Structure and Assessment

At both Bath and Cardiff Met, we encourage our programming apprentices to develop an interdisciplinary mindset and transferability of skills: we view programming as sharing with mathematical thinking in the ways in which we might approach solving a problem; with engineering thinking in ways of designing and evaluating a large, complex systems and with scientific thinking in ways of understanding computability, intelligence, the mind and human behaviour. All of these are fundamental aims of our undergraduate degree programmes. Thus, more specifically, we explicitly try and develop higher-level computational thinking and problem solving skills before a significant focus on syntactic and semantic programming structures. Whilst this abstraction can be conceptually difficult for many students transitioning from secondary education through to university, our courses provide an opportunity to embed this at the start of their degree, intersecting across a number of the key anchor modules. The wide range of experience amongst our students at the point of entry into our programming course means that each of them experiences these transitions differently. Though each individual can be considered a programming apprentice, they encounter the irreversible transformations and moments of conceptual integration at different moments and struggle to different extents with the integration of scientific, engineering and mathematical thinking. These disparities of experience and pace of transformation have direct impact on the ways that we structure our courses.

At Bath, we teach programming as a practical apprenticeship, combining hands-on application with a strong theoretical underpinning. In the light of this twin focus, Bath–M's programming is taught as a concurrent half of an all-year module, resourced as an extremely hands-on unit i.e. with every student getting one hour of *supervised* laboratory time per week. This actually meant that there was twice as much practical time in all as in the previous regime of all of a one-semester module. Having taught programming at both the "normal" (at least for Bath) rate of two hours of lectures and one of laboratory and at this "parity" rate of one hour of programming lecture and one hour of laboratory per week, JHD is convinced that the parity rate is more effective. By way of comparison, Bath–C, where Programming is a double (24 CAT/12 ECTS credits) module, provides three hours of lectures and two of laboratories/week: essentially a halfway house.

Another strategic decision in strong alignment with the apprenticeship model is the assessment method for the course. Bath–C has a fairly traditional view: 1/6 on weekly exercise sheets, 1/3 on major coursework and 1/2 on the written exam. The high-level view of the assessment in Bath–M is also "50% examination, 50% coursework". The detailed breakdown is slightly more subtle: the mathematics component is 38% examination and 12% class test (with computers), while the programming component is 38% coursework and 12% examination. As of writing (November 2015) the course team is considering changes to the detailed breakdown, but the 50:50 split will remain. It is worthwhile considering the Learning Outcomes of the module (from the Catalogue, our numbering): After taking this unit, the student should be able to:

1) *Apply the basic principles of programming in studying problems in discrete mathematics.*
2) *Make proper use of data structures in the applications context.*
3) *Demonstrate understanding of a range of mathematical topics which relate to computation, such as modular arithmetic, elementary graph theory and elementary computational number theory and their applications.*
4) *Analyse the complexity of simple algorithms.*
5) *Explain the use of some famous algorithms such as the Fast Fourier Transform.*
6) *Use the MATLAB programming environment.*

Items 1, 2 and 6 are directly practical skills whilst items 3, 4 and 5 reflect the (complementary) theoretical aims of the course. Our interest in the former (the practical application of programming skills) leads to the 50% coursework

assessment mentioned above, and relates to JHD's opening statement to the course: "*My aim is to show you how to program, my and the laboratory tutors' aim is to help you with your programming, and your aim ought to be to learn programming by doing it*". JHD has used this statement since 2009, but, since encountering [**?**] he has realised that this is really an apprenticeship model, with the students as apprentices, the tutors as journeymen, and the lecturer as the master craftsman. There is a strong emphasis on "learning by doing". For example, in the first lecture JHD writes in front of them (and definitely does not produce a pre-written one out of the hat) a recursive factorial program, and the first exercise is to adapt this into a Fibonacci number program. In fact, this "writing" process is iterative.

The laboratory sessions take place in five one-hour sessions at the end of the week (after the lectures and formal problem classes). They are held in 75-seater laboratories (arranged as five rows of 15 machines each), and the students are allocated to a specific row of 15 within that. Similarly, there are five tutors, each assigned to a specific row. Ideally, the same tutor stays with the same group of 15 all year. This firm allocation of students to tutors, and the briefing to tutors that they are responsible for the learning of their allocated students, means that tutors do occasionally raise concerns about specific students with me in a way that tutors in a floating pool would not. The allocation of students to specific tutors allows the latter to develop a deeper understanding of the experience that former bring to the course. It also allows tutors to anticipate the degree to which early lab exercises present a substantial challenge to individual students and adapt the assistance they provide accordingly.

It has been the authors' gut feeling for many years that weekly practice, and frequent assessment, are important in getting students into the habit of programming. This has recently been borne out by [**?**], who state "Students with high absolute submission counts during tutorials tend to significantly more often get a good grade from the course than those who have low absolute submission counts."

4. Student Engagement

Beyond our interest in the provision of sufficient opportunity for hands on coding, however, we have also worked to foster student engagement with our practical/theoretical apprenticeship model. It is all very well having weekly laboratory sessions, and exercises: how, in a university context, does one make students use them? Bath-C assigns marks for them (1/6 of the total). In Bath-M, where there are no actual marks, the answer is that, in the weeks where there is no formal coursework being done, and especially in the first few weeks, we still set formative weekly exercises. These are graded pass/fail, and are known as 'Tickables', since the laboratory tutor ticks them in the course of each laboratory session. The tutors are instructed that a certain amount of 'coaching' is permissible here (where it would not be in assessed coursework) and that one of the aims, again particularly in the first weeks, is to instil confidence in the students. These exercises are sequential, building both on the lectures and on previous ones. The first Tickable builds on the factorial program written in lectures (as shown above), and asks the students, based on the lecturer's program (which is supplied in the Learning Environment after the lecture), to write a Fibonacci number program, i.e. $F(n) = F(n-1) + F(n-2)$ with suitable base conditions. The students are also asked to reason about the running time of this program, which builds on the discrete mathematics component of the course. The second Tickable extends this to the matrix formulation, and so on. Each subsequent week, the students can build on their solution from the previous week, or on the lecturer's solution. Simple pass/fail marking by the tutor seems easy from the lecturer's point of view, but the experienced academic will ask about:

- appeals from the tutor's decision not to tick;
- the incentives for the students to do the work;
- illnesses and other absences, and requests for extensions, which get in the way of providing the solution in a timely manner for next week's Tickable.

The Bath–M solution to these, potentially very important, questions are as follows:

1) JHD attends all the laboratories for the first Tickable, and resolves any queries on the spot (also helping the new tutors and giving them confidence), possibly announcing any adjudications if this seems appropriate. Thereafter, over a period of six years with 250 students/year doing ten more Tickables each, there has not been a single appeal against the justice of ticking. (There are appeals against the misrecording of ticks, and the tutors can be fallible here, and forget to copy a hand-written record into the Virtual Learning Environment.)
2) The incentive is that, if the student does not get 80% of the ticks (of which there are generally 15) for the course, then the assessed coursework mark is reduced pro rata. In practice this is a rare event, as nearly all students do get over 80%, and those that undershoot do so drastically, fail the coursework anyway, and are generally those students who do not engage at all with the course. What does occasionally happen is that a student misses the first two laboratories (which the Ticking process will record), is reported to the Personal Tutor (the Department Office provides a list of Personal Tutors arranged by student computer username, which helps!) and then starts attending seriously, at which point the missing ticks are condoned.
3) Minor illnesses and absences are explicitly (this is explained as part of the course briefing, and repeated when requests are made) allowed for by the 80% rule, in that a student can miss three without penalty. More would indicate a more serious condition, which should be addressed programme-wide rather than just in this unit. One case that this does not cover is that of sport players who may miss several laboratories, but this is known in advance, and the rule is "submit in advance to your tutor by e-mail". In particular, no extensions are

given for Tickables (unlike assessed coursework), and so the weekly rhythm of "lecturer demonstrates; student work a similar example; lecturer shows his solution, uses it to lead into next demonstration" is not disrupted.

5. Impact on Tutoring: Journeymen

Just as in a traditional craft, the journeymen have also to learn, and to be managed. JHD saw the importance of this whilst at the University of Waterloo, Canada, where the programming lecturer was supported by an entire cast of ISAs (Instructional Support Assistants), IAs (Instructional Apprentices), ISC (Instructional Support Coordinator: essentially a full-time role recruiting and rostering the ISAs and IAs). The tutors at Bath are generally PhD students from the two departments of Mathematical Sciences and Computer Science. One tutor in each department is formally designated as the Senior Tutor, with two sets of responsibilities:

- to the lecturer: to inform them when things are going wrong, and to ensure that the lecturer is aware of generic difficulties;
- to the other tutors, to act as a less formal source of advice and support than going to the lecturer.

These two Senior Tutors, and other experienced tutors, are consulted by the lecturer about changes, and often used to test new assignments or exercises. It should be noted that three of the seven authors of [**?**] are/were Senior Tutors. JHD is often asked by tutors to write references for their teaching abilities as they move on to other jobs, and gladly provide them. One tutor returned to a lecturing post in her native Thailand and occasionally asks JHD questions of teaching practice, as well as using some of the ideas (such as the 80% rule) in her own university. It is important to remember a simple piece of arithmetic: in a given week the lecturer delivers about 1.5 hours of teaching (lecture plus half a problem class), while the laboratory tutors deliver 25 between them. Hence time spent writing briefing material for the tutors is often at least as valuable as time spent preparing lectures. Every exercise is issued with some support material for the tutors: sometimes "what to explain" and sometimes "what not to explain – they have to figure that one out". This is necessary to ensure a uniform experience across the 25 tutorial groups. On a more immediate level, we ensure that at least one, and generally two, of the five tutors in the lab for any hour are those who have done this tutoring in previous years. This is particularly important at the start of the year, when new tutors may not be that familiar with MATLAB, and have yet to learn to spot the baffling blunders that beginners and near-beginners can make.

6. Suggestions for Others

As we have seen, there are significant benefits to the apprenticeship model and approach. This has been highlighted in both the Bath and Cardiff Met cases, across two different departments and discipline areas, but focusing on the teaching of introductory programming. The potential of early development of computational thinking and transferable problem solving skills is clear, as well as fostering a culture of software carpentry and codemanship: developing useful and usable software artefacts. While this is an under-explored area from a pedagogic research perspective, we have seen the benefits from multiple cohorts at the two institutions. Many large-scale programming classes will rely heavily on tutors. They form an important part of the students' learning experience. The following points have been found useful at Bath, where we have large laboratories (efficient for timetabling, but harder to manage):

- Assign students to tutors, rather than having a 'floating pool'. We have certainly been helped here at Bath by having one large laboratory for 75=5*15, rather than smaller laboratories, as there can then be a mix of tutor experience in the room.
- Brief the tutors on what they are expected to do, especially the role in helping (or not) students with weekly exercises versus the assessed coursework.
- Active and engaged tutors are key: ideally they should not sit in labs waiting to be asked for help!
- Debrief the tutors: they know far better than the lecturer what is actually going on in the laboratory classes and with the students, especially if the first point is followed.
- Staff development of tutors is important, and pays off.
- Encourage students to help each other, such as peer support in labs (often but not always more advanced students helping those with less experience) and online, for example on Moodle or other virtual learning environment.

We also have the following general suggestions.

- Find a way (the Bath–M way is only one option, and Bath–C uses a more traditional way) of ensuring weekly exercises are taken seriously by the students; engagement is key, especially for the apprenticeship model to work. This means that the exercises require rapid turnround, ideally automated marking. We note that increased automation was welcomed by the Bath–C students (Staff-Student Liaison Committee 29 October 2015).
- Since the aim is to teach the craft of programming, the lecturer should demonstrate programming, rather than just talk about pre-written programs; furthermore, they should emphasise that their solution is one of perhaps a number of 'correct' solutions, but may not be the optimal solution.
- The choice of programming language should suit the audience and the pedagogic goals, not some pre-defined idea of a 'good' language; be wary of jumping to new (and potentially faddish) languages, tools and environments.
- Develop (and emphasise) computational thinking skills from the start, linking theoretical skills and understanding to real-world problem solving. A wide range of resources [https://goo.gl/dZ6vyj] already exists that can be easily adapted and adopted.

- The value of developing a culture around (and appreciation of) software carpentry and codemanship: essentially, creating useful and usable software artefacts.

This paper has focused on the teaching of introductory programming as a craft skill, to be taught via an apprentice model, with further long term aims of fostering a culture of software carpentry and codemanship: developing useful and usable software artefacts. Teaching it this way has substantial advantages in terms of student engagement, participation and increased retention. There are other craft skills in computing (for example, database design, advanced object-oriented programming, etc.) which could easily benefit from the same approach.

7. Conclusion

We have seen – and will to continue to see – significant changes to computer science education in the UK, from primary school through to FE and HE [**?**]. From reform of the computing curriculum in England, scaling and CPD challenges in Scotland, as well as expected future reform in Wales and Northern Ireland, the curriculum and qualifications landscape will most likely increase the diversity of qualifications and experiences of entrants to HE in the short and medium term. A renewed focus on pedagogy for teaching computer science and programming should be embraced; for example, the formation of the UK Forum for Computing Education [http://ukforce.org.uk], led by the Royal Academy of Engineering and supported by BCS, The Chartered Institute for IT, is a positive step to bring together key stakeholders in the UK.

Furthermore, as a discipline, we are currently in the midst of significant scrutiny, from both an educational, economic and policy perspective; this presents both opportunities and warnings: the opportunity to raise the profile and wider public perception of the discipline as an educational and economically valuable pursuit versus being driven to support the immediate and somewhat transient demands on the technology sectors. Recent and ongoing reviews that will have an impact on our discipline include: the Shadbolt and Wakeham reviews of computer science and STEM graduate employability (as well as degree accreditation); the new UK QAA Subject Benchmark Statement for Computing to be published in early 2016; the 2014 UK Digital Skills Taskforce report [http://www.ukdigitalskills.com] ("*Digital Skills for Tomorrow's World*"), the 2015 UK House of Lords Select Committee on Digital Skills' report [http://www.parliament.uk/digital-skills-committee] ("*Make or Break: The UK's Digital Future*"), as well as emergent effects from the recent or upcoming changes to the computing curricula across the four nations of the UK.

Finally, we acknowledge the impact of the application of computational techniques across science and engineering and how this has fundamentally affected practices within those disciplines. Computing is both a rigorous academic discipline in its own right and also facilitates and supports a wide range of other disciplines, from computational physics to computational social science; in essence it has become a bridge for interdisciplinarity: it now does not only support how science is done, but what science is done [**?**]. Therefore, aspects from this report (and from across computer science) could also be applied to a number of STEM disciplines that now need to teach introductory programming (and how to leverage data and computation to solve domain problems) in their undergraduate curricula.

Acknowledgments

The authors would like to thank all the tutors who have contributed to the development of these courses at both Bath and Cardiff Met: in many ways they are the real teachers.

References

[1] Astrachan, O., and Reed, D., AAA and CS 1: the applied apprenticeship approach to CS 1. ACM SIGCSE Bulletin, 27(1995), 1–5.

[2] Ben-Ari, M., 'Constructivism in Computer Science Education', J. Computers in Mathematics and Science Teaching 20(2001), 45–73.

[3] Boustedt, J., Eckerdal, A., McCartney, R., Moström, J. E., Ratcliffe, M., Sanders, K. and Zander, C., 'Threshold Concepts in Computer Science: Do They Exist and Are They Useful?', Proc. 38th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'07), ACM Press, pp. 504–508.

[4] Brown, N.C.C., Kölling, M., Crick, T., Peyton Jones, S., Humphreys, S., and Sentance, S. "Bringing Computer Science Back Into Schools: Lessons from the UK,", Proc. 44th ACM Technical Symposium on Computer Science Education (SIGCSE'13), ACM Press, pp. 269–274.

[5] Brown, N.C.C., Sentance, S., Crick, T., and Humphreys, S. "Restart: The Resurgence of Computer Science in UK Schools," *ACM TOCE* 14(2), pp. 1–22, 2014.

[6] Clancy, M., Computer Science Education Research, Routledge, chapter Misconceptions and Attitudes that Interfere with Learning to Program, pp. 85–100, 2004.

[7] Davenport, J.H., Wilson, D., Graham, I., Sankaran, G., Spence, A., Blake, J. and Kynaston, S., Interdisciplinary Teaching of Computing to Mathematics Students: Programming and Discrete Mathematics. To appear in *MSOR Connections*. http://journals.heacademy.ac.uk/doi/abs/10.11120/msor.2014.00021. http://opus.bath.ac.uk/37841/.

[8] Eckerdal, A., McCartney, R., Moström, J. E., Ratcliffe, M., Sanders, K. and Zander, C., 'Putting Threshold Concepts into Context in Computer Science Education', Proc. 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITICSE'06), ACM Press, pp. 103–107.

[9] Hazzan, O., Lapidot, T. and Ragonis, N., Guide to Teaching Computer Science: An Activity-Based Approach, Springer, 2011.

[10] Khalife, J., 'Threshold for the introduction of programming: providing learners with a simple computer model', in Proc. 28th Int. Conf. on Information Technology Interfaces, pp. 71–76, 2006.

[11] Land, R., Meyer, J. H. and Smith, J., eds, Threshold Concepts and Transformational Learning, Vol. 16 of Educational Futures: Rethinking Theory and Practice, Sense Publishers, 2008.

[12] Meyer, J. H., Land, R. and Baillie, C., eds, Threshold Concepts and Transformational Learning, Vol. 42 of Educational Futures: Rethinking Theory and Practice, Sense Publishers, 2010.

[13] Royal Society, 'Science as an open enterprise', https://royalsociety.org/topics-policy/projects/science-public-enterprise/report.

[14] Vihavainen, A., Paksula, M. and Luukkainen, M., 'Extreme Apprenticeship Method in Teaching Programming for Beginners', in Proc. 42nd ACM Technical Symposium on Computer Science Education (SIGCSE'11), ACM Press, pp. 93–98.

[15] Willman, S., Lindéna, R., Kaila, E., Rajala, T., Laakso, M.-J. and Salakoski, T., 'On study habits on an introductory course on programming', Computer Science Education 25(3) pp. 276–291, 2015.